Vector Based Rotation Invariant Convolutional Filters Involving Aerial Imagery

Zachary Varnum
University of Connecticut School of Engineering
Storrs, Connecticut, United States
zachary.varnum@uconn.edu

Daniel Gove
University of Connecticut School of Engineering
Storrs, Connecticut, United States
daniel.l.gove@uconn.edu

Nikolas Anagnostou
University of Connecticut School of Engineering
Storrs, Connecticut, United States
nikolas.anagnostou@uconn.edu

I. Introduction

Aerial imagery can be applied in many fields. Some of these include urban planning, agriculture, environmental monitoring, and disaster response. However getting meaningful information out of these images is difficult specifically in scenarios where objects have scale, rotation, and viewpoint variations. Traditional convolutional neural networks (CNNs) work well in plenty of image recognition tasks but can struggle with these variation challenges quite a bit. This can lead to suboptimal performance in object detection on aerial imagery.

We will mainly focus on making rotation invariant convolutional filters for object detection tasks in aerial imagery. We plan to improve the accuracy of object detection systems through the rotation invariance. This can help with more quality analysis and interpretation of aerial imagery.

The significance of our project is that it has the potential to advance the capabilities of computer vision systems in handling diverse and complex aerial imagery. By incorporating rotation invariance directly into the convolutional filters, we plan to simplify the pipeline and process for object detection in aerial imagery.

Through this project, we also plan to possibly contribute to the development of more sophisticated and adaptable computer vision algorithms that can better support applications such as urban planning, agriculture, environmental monitoring, and disaster response efforts.

II. Background and Related Work

A. Rotation Equivariant Vector Field Networks

One paper we researched to aid us with making our project is Rotation Equivariant Vector Field Networks [1].

Traditional convolutional neural networks (CNNs) are translation equivariant. This means that they can recognize patterns regardless of the position in the input image. However a downside is that they lack equivariance to other transformations like rotation.

This rotation equivariance is very well suited for things like object recognition and pose estimation. In both of these tasks, objects may appear in various orientations.

Making it so one can represent rotation-equivariant features is challenging. Standard CNNs are not necessarily suited for this task so it can be quite difficult. They require a lot of data augmentation and/or other post processing steps to handle the rotation.

The big motivation behind RFVN (Rotation Equivariant Vector Field Networks) is needing proper neural networks that can model rotation equivariance correctly. The paper lists limitations of previous approaches in capturing these rotation equivariant features.

Some key components of RFVN include group convolutions, vector field representation, and rotation-equivariant pooling. The paper studies RFVN on plenty of different tasks such as synthetic benchmarks and real world applications. It shows better performance compared to the previous methods. RFVN can also be improved with modifications to aspects like the network architecture, optimization techniques and integration with other models.

RVFN seems like it produces promising results but it still may encounter challenges with more complex transformations and scalability for larger datasets and high dimension inputs. For future research, there can be a focus on improving efficiency and effectiveness and exploring applications elsewhere.

RFVN can impact a lot of fields where these rotation equivariant representations are important such as computer vision, robotics, and scientific computing. It opens up new possibilities for applications that need robustness for transformations including rotation.

B. SSD: Single Shot Multibox Detector

Another paper that we explored to reach our proposed method is SSD: Single Shot Multibox Detector [2].

Before SSD, object detection mainly relied on region proposal-based methods like R-CNN (Region-based Convolutional Neural Networks). Different variants also exist such as fast R-CNN and faster R-CNN. All of these different methods consisted of a two-stage process of region proposal generation and also classification. On the contrary SSD performs both localization and classification in a single feed-forward pass.

SSD highlights limitations and downsides of region proposal-based methods. It addresses the speed and efficiency of these methods. It gets rid of the need for a

separate region proposal step. This makes it so that SSD can achieve real time performance.

The paper also outlines the importance of efficiency in object detection for different applications. These can include autonomous driving, robotics, and surveillance.

These are some key components of SSD:

Multi-scale feature maps - SSD makes use of feature maps of different scales. This is so it can detect objects of different sizes. Feature maps can provide plenty of contextual information.

Default boxes - SSD predicts object bounding boxes. To do this, it uses a set of default boxes with different aspect ratios at each spatial location of the feature maps.

Prediction modules - SSD also makes use of many convolutional layers to predict class scores and adjust bounding box coordinates for each default box across multiple scales.

The paper also talks about evaluating SSD on benchmark datasets. Some of these include Pascal VOC and COCO, which shows a competitive performance as far as speed and accuracy are concerned (comparing it to existing methods). SSD attains state-of-the-art results in real-time object detection. Again, this is able to surpass methods that came before it regarding both accuracy and efficiency.

Since SDD was released to the public it has inspired a lot of extensions and variants, with the goal of improving the performance and adapting it to specific applications. Some examples of these variants include SSD with different backbone architectures, which are SSD MobileNet and SSD ResNet. These both trade off speed and accuracy based on specific deployment requirements.

SSD definitely achieves impressive and notable results regarding real time object detection. However it can still face plenty of challenges in detecting small objects, and also in dealing with obstructions.

Some future research directions could be able to focus on improving the robustness of SSD in scenarios where it doesn't do the best. There can also be research in other techniques for reducing false positives and improving accuracy.

SSD has a grand impact regarding the field of computer vision. This is particularly true for applications requiring real time object detection.

Again, the efficient architecture of SDD and competitive performance have made it a good choice for many practical applications, including surveillance, traffic monitoring, and object tracking.

III. Methods

In our architecture, we explore an approach to achieving rotation invariance in convolutional neural networks through vector representations of the convolutional filters and responses.

By utilizing both Cartesian and polar forms of vectors, our method leverages the inherent properties of these coordinate systems—vector addition in Cartesian and rotation-scaling in polar—to effectively handle orientations in image data.

Our goal is to represent the response of a filter and the feature's prominent direction using vectors. There are two ways to represent a vector:

Polar Form (r, θ) ia good for changing size and direction

Cartesian Form (x, y) is good for adding and subtracting vectors

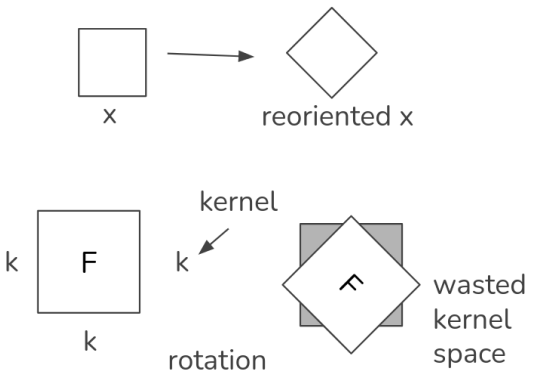
Say we have an image x and a convolutional filter F and we want to measure how well F fits each region of x with one caveat; x can be oriented in an arbitrary direction.

Regular convolutional filters are sensitive to orientation. It could be that F only matches a region when x is in a very particular orientation.

To recognize x at any of its orientations, we rotate F by some degree θ, $F = F \cdot R\theta$, and if we iteratively search through θ from 0 to 360 degrees, we find all the ways that F fits x at different orientations. Searching θ has the potential to help prevent overfitting and force the model not to make assumptions about the orientation of features. In Aerial Imagery, the orientation of features on the ground is totally random, since there is no reason that all objects on earth should be facing a certain way, or that devices taking aerial photography need to be oriented in a certain direction, which is why we believe that regularizing rotations is important. As we explored this approach we identified several issues and challenges.

The first challenge was finding a way to generate several rotation matrices and how to apply them to the input in pytorch. We wrote a function parameterized by the number of rotations we desire and returned a tensor of rotation matrices that would be applied to each filter, so that we had several copies of each filter at different rotations.

The next issue with this we identified was that while rotating the filters, their corners would leave the kernel space leading to several death parameters, and the corner's of the kernel would be empty, so we were undermining the efficiency of our model.



We identified a few ways to potentially make the θ scan more efficient. A 4-way scan ensures full usage of the kernel space and requires no interpolation or extra memory. To achieve 90 degree rotations we simply need to change how we traverse the original filter tensor in memory (See the code snippet below)

```
def rotate_filters(self, filters):
  rotated_filters = torch.stack([
    filters,
    filters.transpose(2, 3).flip(2),
    filters.flip(2).flip(3),
    filters.transpose(2, 3).flip(3)
  ], dim=0).view(-1, *filters.shape[1:])
  return rotated_filters
```

The function that parameterized the number of rotations was much more expensive to compute and complicated to develop:

```
def generate_rotation_matrices(self):
  # Generate rotation matrices for the specified number of rotations
  angles = torch.linspace(0, 2 * torch.pi,
                          self.N_rotations + 1, requires_grad=False)[:-1]
  theta = torch.zeros((self.N_rotations, 2, 3))
  theta[:, 0, 0] = theta[:, 1, 1] = torch.cos(angles)
  theta[:, 0, 1] = -torch.sin(angles)
  theta[:, 1, 0] = torch.sin(angles)

  # Use the rotation matrices to create affine grids
  grid = F.affine_grid(theta,
                       torch.Size([self.N_rotations, 1, *self.kernel_size]),
                       align_corners=False)
  return grid
```

However, a coarse 4-way scan can easily miss the majority of feature orientations, and so would require more filters.

Another option we came up with was to use kernel padding. Kernel padding would involve leaving empty space in the kernel to allow F to rotate without any parameters leaving the kernel and becoming dead. One issue with this was that implementation would be a nightmare, which is also true for our third idea, which was to use circular kernels and filters, which could be rotated without worrying about wasted kernel space or parameters.

Ultimately, we decided to use the 4-way scan method in our approach due to its simplicity and efficiency. Another added benefit to the 4-way scan was that the gradients of the filter were easy to propagate through the rotations, since in a 4-way scan there was no need for interpolation.

Once the filters are rotated and we convolve them over the input, we are left with several feature maps for each filter, with each feature map corresponding to a particular filter and rotation angle θ.

In order to explicitly capture the angle θ in the feature maps themselves, we constructed a tensor full of the appropriate θ and stacked it with the feature map, so that the last dimension was of the form [Response, θ].

Then, for every filter's set of augmented feature maps, we wanted to aggregate the results into one channel in an attempt to capture the prominent feature orientation. We identified two ways of doing this, the first is to use magnitude based max pooling over the feature maps, where we drop the responses for all but the filter orientation that produced the highest response, and the other was summation, where the [Response, θ] pairs, treated as polar vectors, were converted into cartesian coordinates and summed, producing a composition of all the responses.

```
if self.aggregation == "sum":
  # Convert into cartesian coordinates
  x_conv = output * torch.cos(output_phase)
  y_conv = output * torch.sin(output_phase)

  # Add the outputs in cartesian coordinates
  output_x = torch.sum(x_conv, dim=2)
  output_y = torch.sum(y_conv, dim=2)

  # Convert the results to polar coordinates
  output_mag = torch.sqrt(output_x**2 + output_y**2)
  output_phase = torch.atan2(output_y, output_x)

  vectorized_output = torch.stack([output_mag, output_phase], dim=-1)
else:
  # Apply Max Pooling, drop excess rotation channels
  max_magnitude, max_indices = torch.max(output_mag, dim=2, keepdim=True)
  max_phase = torch.gather(output_phase, 2, max_indices)

  vectorized_output = torch.stack([max_magnitude.squeeze(2), max_phase.squeeze(2)], dim=-1)
```

After the aggregation of the responses we are left with a feature map of polar vectors. In order to introduce non-linearity we needed to define an activation function that could work on these features maps.

We decided that a good activation function would be a relu that operates only on the strength of the response while leaving the angle alone, since we do not want to have the network treat some orientations differently.

```
class VectorRelu(nn.Module):
  def __init__(self):
    super(VectorRelu, self).__init__()

  def forward(self, x):
    magnitude = x[..., 0]
    phase = x[..., 0]
    magnitude = F.relu(magnitude - 1) + 1
    return torch.stack([magnitude, phase], dim=-1)
```

Since after summing and conversions to and from cartesian coordinates the magnitude of a vector will be positive, we chose to 0 the response if it was less than 1, instead of 0.

In order to build more complex feature maps we need to be able to stack multiply convolutional filters on top of each other, so we needed to define a type of convolutional layer that can operate on these vectorized feature maps.

In these special convolutional layers, the filters themselves are composed of vectors and they operate very similarly to the previous example.

When multiplying two vectors together, we multiply their magnitudes and add their phases and, for the convolution, they are then summed in cartesian coordinates.

As an example, for a set of input filters x and a filter F, if

x = (r1, θ1), (r2, θ2), (r3, θ3)

and F = (fr1, fθ1), (fr2, fθ2), (fr3, fθ3),

then F*x = (r1 · fr1, θ1 + fθ1) + (r2 · fr2, θ2 + fθ2) + (r3 · fr3, θ3 + fθ3)

The filters are rotated in the same way, and are also aggregated before being returned.

The only building block we have left is MaxPooling, which can be achieved for vectors using the same method we used for the ReLU where we ignore the phase:

```python
class VectorMaxPool2d(nn.Module):
  def __init__(self, kernel_size, stride=None, padding=0):
    super(VectorMaxPool2d, self).__init__()
    # Initialize parameters similar to nn.MaxPool2d
    self.kernel_size = kernel_size
    self.stride = stride if stride is not None else kernel_size
    self.padding = padding

  def forward(self, x):
    # Extract magnitude and phase
    magnitude = x[..., 0]
    phase = x[..., 1]

    # Apply max pooling to the magnitude part
    pooled_magnitude, indices = F.max_pool2d(
        magnitude,
        self.kernel_size,
        self.stride, self.padding,
        return_indices=True)

    phase_flat = phase.view(phase.shape[0], phase.shape[1], -1)
    pooled_phase = torch.gather(
        phase_flat, 2,
        indices.view(phase.shape[0], phase.shape[1], -1))
    pooled_phase = pooled_phase.view_as(pooled_magnitude)

    pooled_output = torch.stack([pooled_magnitude, pooled_phase], dim=-1)
    return pooled_output
```

What we had hoped to achieve with the vector based convolutional filters was to rotate and scale the vectors in the feature map in such a way that they could either align (stack up) or cancel out (go in opposite directions), as a way to measure complex features.

While it sounds good in theory, there were a number of other issues and challenges that we faced with this approach. The first was that these layers used twice as many trainable parameters (to capture the phase and the magnitude) as traditional convolutional layers, and the gradients were a lot more complicated since they had to be traced all the way through rotation and conversions to and from polar coordinates.

This led to models taking up potentially a significant amount of extra memory than their conventional counterparts and training was much slower.

Our hope was that, despite these drawbacks, this method would outperform conventional methods on aerial imagery due to the fact that the response magnitudes where invariant to rotations of objects and features in the input, since in theory those rotations would be captured by an appropriate filter after that filter had been rotated.

IV. Results

A. Preliminary Results

After building our components it was easy to get started building models and trying to train them. During development for experimentation we used rotated MNIST digits to quickly tinker with the layers and to ensure that they were working as expected.

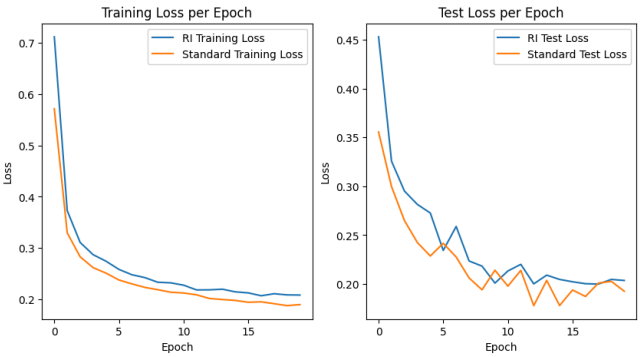
Building a simple model out of our components is very easy:

```python
class ROTNet(nn.Module):
    def __init__(self):
        super(ROTNet, self).__init__()
        # Define the first convolutional layer
        self.vector_transform = VectorTransformConv2d(1, 4, kernel_size=3,
        self.pool1 = VectorMaxPool2d(2, stride=2)
        self.conv2 = VectorConv2d(4, 8, kernel_size=3, aggregation="pool")
        self.vec2mag = Vector2Magnitude()
        self.pool2 = nn.MaxPool2d(2, stride=2)

        self.batch_norm = nn.BatchNorm2d(8)

        self.fc1 = nn.Linear(8*6*6, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = self.vector_transform(x)
        x = self.pool1(x)
        x = vector_relu(x)
        x = self.conv2(x)
        x = self.vec2mag(x)
        x = self.pool2(x)

        x = self.batch_norm(x)

        x = x.view(-1, 8*6*6)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

As a preliminary test we compared this model to a similar model using conventional layers on the rotated MNIST dataset:

```python
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=2)
        self.pool1 = nn.MaxPool2d(2, stride=2)
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3)
        self.pool2 = nn.MaxPool2d(2, stride=2)

        self.batch_norm = nn.BatchNorm2d(8)

        self.fc1 = nn.Linear(8*6*6, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(self.pool1(self.conv1(x)))
        x = F.relu(self.pool2(self.conv2(x)))

        x = self.batch_norm(x)
        x = x.view(-1, 8*6*6)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

The results were as follows:



The preliminary results do not suggest anything exceptional about our methodology.

To get to this point, and before we could train on our dataset of Aerial Imagery, we faced a number of challenges. At first we noticed that our model did not converge and maintained a loss of 2.3, indicating that it was making random choices, and this could not be remedied by lowering the learning rate.

Upon close investigation it was found that the model's gradients were very unstable, where some partials routinely reached over 1000 when others dropped to 0.1 or 0.01.

```
vector_transform.filters, Gradient: 24.42447853088379
vector_transform.bias, Gradient: 4.794795036315918
conv2.filter_x, Gradient: 3.6289360523223877
conv2.filter_y, Gradient: 3.367757558822632
conv2.bias, Gradient: 0.037144843488931656
fc1.weight, Gradient: 707.3741455078125
fc1.bias, Gradient: 0.266198068857193
fc2.weight, Gradient: 148.81219482421875
fc2.bias, Gradient: 0.41629207134246826
Train Epoch: 1 [0/60000 (0%)]   Loss: 44.110535
Train Epoch: 1 [6400/60000 (11%)]       Loss: 2.299814
Train Epoch: 1 [12800/60000 (21%)]      Loss: 2.312032
Train Epoch: 1 [19200/60000 (32%)]      Loss: 2.302313
Train Epoch: 1 [25600/60000 (43%)]      Loss: 2.302266
Train Epoch: 1 [32000/60000 (53%)]      Loss: 2.310998
Train Epoch: 1 [38400/60000 (64%)]      Loss: 2.301167
Train Epoch: 1 [44800/60000 (75%)]      Loss: 2.294712
Train Epoch: 1 [51200/60000 (85%)]      Loss: 2.298670
Train Epoch: 1 [57600/60000 (96%)]      Loss: 2.294280

Test set: Average loss: 2.3015, Accuracy: 1135/10000 (11%)
```

After reflecting on this problem and trying a few different solutions such as different ways to initialize the weights, we found that batch normalization helped the gradients remain healthy throughout the model and we achieved convergence:

```
vector_transform.filters, Gradient: 0.12126896530389786
vector_transform.bias, Gradient: 0.024341803044080734
conv2.filter_x, Gradient: 0.06731051951646805
conv2.filter_y, Gradient: 0.059551406651735306
conv2.bias, Gradient: 0.0004796787688974291
batch_norm.weight, Gradient: 0.6434422135353088
batch_norm.bias, Gradient: 0.3902709186077118
fc1.weight, Gradient: 0.10934989154338837
fc1.bias, Gradient: 0.045144613832235336
fc2.weight, Gradient: 0.3736788332462311
fc2.bias, Gradient: 0.04259234294295311
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.363023
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.698534
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.432561
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.478418
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.175316
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.317137
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.459591
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.242937
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.257109
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.321476

Test set: Average loss: 0.0890, Accuracy: 9718/10000 (97%)
```

B. Object Detection in Aerial Imagery

The dataset we initially used was the NWPU VHR-10 dataset [3]. The NWPU VHR-10 dataset contains 800 very high resolution satellite images and annotations for bounding boxes around classified objects such as planes, harbors, baseball fields, and more.

The labels for each image came as a text file containing the bounding boxes: (x1, y1), (x2, y2), and an object class (1-10). In order to build a model to make these kinds of detections, we investigated the Single Shot Multibox Detector (SSD), an approach known for its simplicity and effectiveness in object detection [2]. However, we found that models built using SSD were typically large and resource-intensive, which was not feasible with our limited computational capabilities.

This exploration led us to a critical aspect of SSD: the necessity to pad label data to the maximum number of detections and include confidence scores to account for potential non-detections. These scores help manage the sparsity of real detections versus potential false positives or negatives, crucial for training robust detection models.

Constructing the loss function for our model proved challenging. We developed a custom data loader and a transformation pipeline which included a tailored loss function to facilitate model training. Despite these efforts, our model struggled to converge; it progressively lowered its confidence scores, effectively predicting zero objects across evaluations—a phenomenon reflecting in the high loss values. We failed to get even a conventional CNN to converge and we believe that this was in large part due to our inability to accommodate larger, more complex models.

```python
class MrEngineerMan(nn.Module):
    def __init__(self, img_height=384, img_width=512, num_boxes=67, num_cla
        super(MrEngineerMan, self).__init__()
        self.num_boxes = num_boxes
        self.num_classes = num_classes
        self.img_height = img_height
        self.img_width = img_width

        # Standard Convolutional Layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        # For better gradients
        self.batch_norm1 = nn.BatchNorm2d(32)
        self.batch_norm2 = nn.BatchNorm2d(64)
        self.batch_norm3 = nn.BatchNorm2d(128)

        # Calculate the output size after convolutions and pooling
        out_size = self.calculate_conv_output_size()

        # Boundary Box Prediction
        self.fc1_bbox = nn.Linear(out_size, 256)
        self.fc2_bbox = nn.Linear(256, num_boxes * 4)

        # Class Prediction
        self.fc1_class = nn.Linear(out_size, 256)
        self.fc2_class = nn.Linear(256, num_boxes * num_classes)

        # Confidence score prediction
        self.fc1_conf = nn.Linear(out_size, 256)
        self.fc2_conf = nn.Linear(256, num_boxes)
```

Several adjustments were made to address these issues, including modifying the confidence weighting in the loss function, applying various normalization techniques, and segmenting the decision heads for confidence, bounding box regression, and class predictions. Nevertheless, these modifications underscored a stark reality: our computational resources were insufficient for the complexity of tasks we aimed to perform.

To align better with our capabilities, we shifted focus to a simpler task—image classification—using the EuroSAT dataset. This transition was accompanied by adopting a new, yet larger, architectural model that maximized our available memory and required extensive training time. The reduction in filter numbers in our custom network model was a necessary compromise to fit our memory constraints.

The EuroSAT dataset comprises 27,000 labeled and geo-referenced satellite images from the Sentinel-2 mission, covering 13 spectral bands and distributed across 10 different land use and land cover categories. This dataset is designed to support the development and benchmarking of machine learning models for land classification tasks [4].

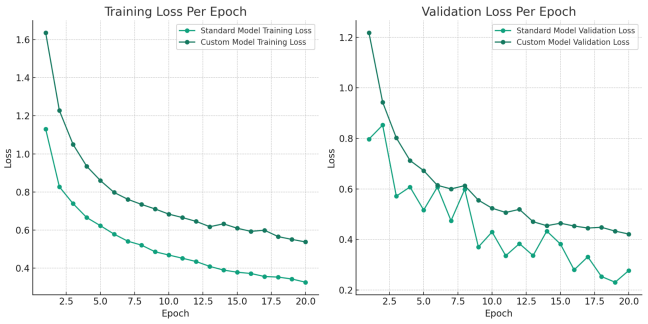
For the EuroSAT classification task we used the following model:

```
class MrModel(nn.Module):
    def __init__(self, num_classes=10):
        super(MrModel, self).__init__()
        # Define the convolutional block
        self.features = nn.Sequential(
            self.conv_block(3, 32),
            self.conv_block(32, 64),
            self.conv_block(64, 128),
            self.conv_block(128, 256),
            nn.AdaptiveAvgPool2d((1, 1))
        )

        # Define the classifier block
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 10)
        )

    def conv_block(self, in_channels, out_chan
        return nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.MaxPool2d(2, 1)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

The model was adapted to use our vectorized filters as follows:

```
class MrSpecialModel(nn.Module):
    def __init__(self, num_classes=10):
        super(MrSpecialModel, self).__init__()
        # Define the convolutional block
        self.features = nn.Sequential(
            self.vector_transformation_block(3,
            self.vector_conv_block(16, 32),
            self.vector_conv_block(32, 64),
            self.vector_conv_block(64, 128),
            Vector2Magnitude(),
            nn.AdaptiveAvgPool2d((1, 1))
        )

        # Define the classifier block
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 10)
        )

    def vector_transformation_block(self, in_ch
        return nn.Sequential(
            VectorTransformConv2d(in_channels=i
            VectorBatchNorm2d(out_channels),
            VectorRelu(),
            VectorMaxPool2d(2, 1)
        )

    def vector_conv_block(self, in_channels, ou
        return nn.Sequential(
            VectorConv2d(in_channels=in_channel
            VectorBatchNorm2d(out_channels),
            VectorRelu(),
            VectorMaxPool2d(2, 1)
        )
```

We trained the models over 10 epochs on the EuroSAT dataset classification task using CrossEntropyLoss. The conventional model took 1hr 15min to train, while the vector convolutional model took 4hr 30min to train. After the 10 epochs we achieved the following results:



Due to the memory constraints of the larger model, we needed to use a smaller batch size (16) compared to the (32) batch size on the conventional model, which likely explains the lower volatility in the validation loss of the customized model.

V. Conclusion

Despite these extensive efforts and the iterative adjustments to our model and training approach, the results—graphically represented in the provided plots—suggest that while our model was functional, it did not demonstrate superiority over simpler, similarly-sized models. This project has been a profound learning experience, illustrating not only the complexities inherent in machine learning, particularly in object detection, but also the significant computational demands these algorithms entail even for relatively small datasets and simple problems such as EuroSAT.

References

[1] Marcos, Diego, et al. “Rotation Equivariant Vector Field Networks.” arXiv.Org, 25 Aug. 2017, arxiv.org/abs/1612.09346.

[2] Liu, Wei, et al. “SSD: Single Shot Multibox Detector.” arXiv.Org, 29 Dec. 2016, arxiv.org/abs/1512.02325.

[3] Gong Cheng, Junwei Han, Peicheng Zhou, Lei Guo. "Multi-class geospatial object detection and geographic image classification based on collection of part detectors," published in the ISPRS Journal of Photogrammetry and Remote Sensing, vol. 98, pp. 119-132, 2014.

[4] Helber, Patrick, Bischke, Benjamin, Dengel, Andreas, and Borth, Damian. "Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2019.